3006ICT Robotics and Computer Vision

PRACTICAL BOOK • CHAPTER 9

Vision-Guided Object Picking with KUKA youBot

What you will build: A compact perception-to-action pipeline: the overhead camera observes an orange object, your detector finds its image location, a safety check decides whether the robot is allowed to act, and the provided youBot grasp sequence performs the pick.

Student practical guide

How to use this chapter

This chapter guides you through Workshop 9. It explains the reasoning, key OpenCV and NumPy functions, code structure, debugging checks, and decisions needed to complete the implementation.

Do not try to understand the whole controller first. Your main coding task is to complete detect_orange_block(), verify its output, and then connect the valid result to the supplied grasp helper.

Learning outcomes

- Inspect a Webots youBot environment and identify the camera, arm and gripper components.
- Convert a Webots camera buffer into an OpenCV image.
- Use HSV colour segmentation to isolate an orange object.
- Clean a binary mask using morphological opening and closing.
- Use contours to select a candidate object and reject weak detections.
- Calculate an image centroid using cv2.moments().
- Use image coordinates to implement a simple safety region.
- Separate perception code from supplied robot-motion code.
- Test both a successful case and a controlled failure case.

Workshop structure

Part	Focus	Time
1	Inspect and run the official youBot environment	15 min
2	Detect the orange target	30 min
3	Visualise detection and apply a safe region	20 min
4	Connect vision to the provided grasp	15 min
5	Test one controlled failure	10 min

Part 1: Inspect and Run the youBot Environment

The supplied world is adapted from the official KUKA youBot sample.

The mobile base remains stationary, an overhead RGB camera observes the orange target, and the helper file contains the tested arm and gripper motion. You are therefore focusing on perception and safe action gating, not inverse kinematics or grasp planning.

Establish a baseline first

1. Open worlds/week9_youbot_vision_pick.wbt.
2. Identify the youBot base, five arm joints, two gripper fingers, orange target and overhead camera.
3. Run the starter controller before changing your code.
4. Confirm that the arm reaches HOME and that the robot does not grasp yet.
5. Record the basic timestep and RGB image resolution.

Key Webots setup

```
from controller import Robot

robot = Robot()
timestep = int(robot.getBasicTimeStep())
```

```
camera = robot.getDevice("camera")
camera.enable(timestep)
```

“Robot()” creates the Webots controller object. “getBasicTimeStep()” returns the simulation timestep in milliseconds. “getDevice()” retrieves a Webots device by its exact name, and “enable(timestep)” enables camera sampling.

Debugging checkpoint: If Webots reports a device-name error, check the Scene Tree and the exact device name. Do not guess names or modify the world unnecessarily.

Understand the supplied helper

- “initialise_motion(...)” prepares the arm and gripper devices.
- “go_home(...)” moves the arm to the known HOME configuration.
- “pick_official_target(...)” executes the supplied grasp sequence.
- For the core workshop, do not change the official joint targets or motion timing.

Checkpoint questions

- Why is it useful to test the supplied motion before adding perception?
- Why should the robot remain still when perception returns no valid target?
- Which part of the system is your group responsible for implementing?

Part 2: Detect the Orange Target

Your main implementation is “detect_orange_block(bgr)”. The supplied camera helper already converts the Webots camera buffer into OpenCV BGR format. Build the detector one stage at a time and inspect intermediate results.

The pipeline

Step	Purpose	Useful function
1	BGR → HSV	cv2.cvtColor
2	Keep orange pixels	cv2.inRange
3	Remove small noise	cv2.morphologyEx + MORPH_OPEN
4	Fill small gaps	cv2.morphologyEx + MORPH_CLOSE
5	Find connected regions	cv2.findContours
6	Select strongest candidate	max(..., key=cv2.contourArea)
7	Reject tiny candidate	cv2.contourArea
8	Compute centre	cv2.moments

HSV colour segmentation

Use the threshold values supplied by the workshop.

```
hsv = cv2.cvtColor(bgr, cv2.COLOR_BGR2HSV)

mask = cv2.inRange(
    hsv,
    (H_low, S_low, V_low),
    (H_high, S_high, V_high)
)
```

“cv2.inRange()” creates a binary mask: pixels inside the HSV interval become 255; all others become 0. In this workshop, the provided threshold is intended for the orange target in the supplied world.

What to inspect: Save or display the mask. If the target is not clearly visible as a white region, do not continue to contours yet. Fix the segmentation stage first.

Morphological cleanup

```
kernel = np.ones((5, 5), np.uint8)

mask = cv2.morphologyEx(mask, cv2.MORPH_OPEN, kernel)
mask = cv2.morphologyEx(mask, cv2.MORPH_CLOSE, kernel)
```

Operation	Typical effect	Purpose here
OPEN	Removes small isolated foreground regions	Reduce tiny orange noise
CLOSE	Fills small holes/gaps	Make the target region more connected

The kernel is a small structuring element. The workshop gives you a 5×5 kernel, so use that first rather than tuning parameters unnecessarily.

Find contours

```
contours, hierarchy = cv2.findContours(
    mask,
    cv2.RETR_EXTERNAL,
    cv2.CHAIN_APPROX_SIMPLE
)
```

“cv2.findContours()” returns boundaries of connected foreground regions. “RETR_EXTERNAL” asks for the outermost contours. Always handle the empty-list case: no contour is a valid outcome for a detector.

Select and validate a candidate

Complete the missing return logic yourself.

```
if not contours:
    # return an invalid-detection result

best = max(contours, key=cv2.contourArea)
area = cv2.contourArea(best)

if area < 150:
    # reject the candidate
```

The “key” argument tells Python which value to compare when finding the maximum. Here it compares contour areas. The 150-pixel threshold is a workshop-specific validity check, not a universal object-detection rule.

Compute the centroid

Key mathematical relationship — finish the implementation.

```
M = cv2.moments(contour)

# Check M["m00"] before dividing.
# Then use:
# u = M["m10"] / M["m00"]
# v = M["m01"] / M["m00"]
```

Image moments provide quantities that can be used to estimate the centre of a shape.

“m00” is related to the contour area, while “m10” and “m01” provide the first-order terms. The important programming issue is to check that “m00” is not zero before dividing.

Expected return: A valid detection should provide “(u, v, mask, contour)”. If there is no valid candidate, return “None”. Do not create a fake centre.

Recommended implementation order

1. Make BGR → HSV work.
2. Create the orange mask.
3. Apply opening and closing.
4. Find contours and print how many were found.
5. Select the largest contour and print its area.
6. Apply the 150-pixel rejection test.
7. Calculate “(u, v)” using moments.
8. Return the requested values and test the main controller.

Common mistakes

Symptom	Likely cause	Check
No contour	Threshold/mask problem	Inspect mask image
Many tiny contours	Noise remains	Check morphology and threshold
Centre is wrong	Wrong contour or moments calculation	Draw the selected contour
Crash during centroid	Zero moment	Check “M['m00']”
Robot moves with no target	Invalid result not rejected	Return “None”

Part 3: Visualise Detection and Apply a Safe Region

Once the detector works, you need to make its output understandable. Visualisation is not just presentation: it is a debugging tool that lets you check whether the computer's interpretation agrees with what you see.

Image coordinates

OpenCV image coordinates use the top-left corner as the origin. “u” increases to the right and “v” increases downward.

```
cx = camera.getWidth() // 2
cy = camera.getHeight() // 2
```

Quantity	Meaning
u	Horizontal pixel coordinate
v	Vertical pixel coordinate
(cx, cy)	Centre of the camera image

Draw what your detector believes

Key OpenCV drawing functions. The exact display values are supplied in the workshop.

```
overlay = bgr.copy()

cv2.drawContours(overlay, [contour], -1, ..., 2)
cv2.circle(overlay, (u, v), ..., ...)
cv2.rectangle(overlay, (...), (...), ..., 2)
```

“bgr.copy()” is important: it creates a separate image for annotation, so your original camera image remains unchanged.

“cv2.drawContours()” shows the detected boundary, “cv2.circle()” marks the centre, and “cv2.rectangle()” visualises the safe region.

The safe region

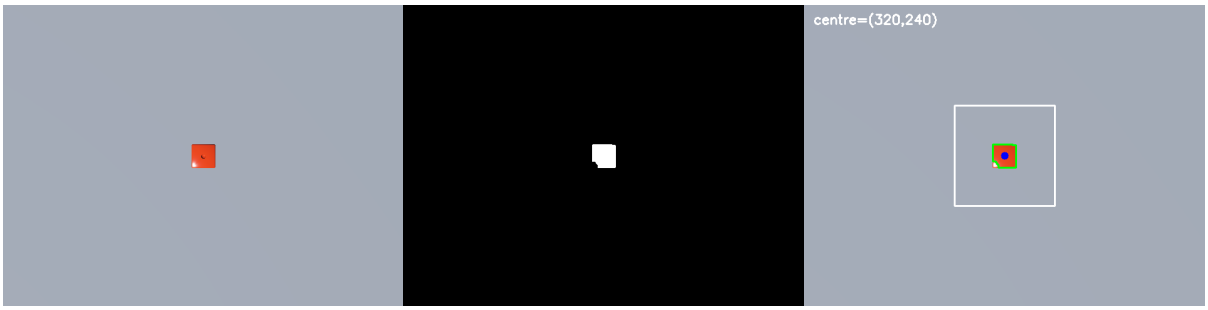
Understand the logic.

```
inside_safe_roi = (
    abs(u - cx) <= 80 and
    abs(v - cy) <= 80
)
```

This is an axis-aligned rectangular region around the image centre. The supplied grasp is calibrated for the official target location, so the detector is being used as a safety gate: the robot is allowed to grasp only when the detected centre is sufficiently close to the expected image location.

Inspect the saved evidence

- “week9_rgb.png” — the original camera view.
- “week9_mask.png” — the binary segmentation result.
- “week9_detection.png” — the annotated detection and safe region.



1. Check that the contour follows the orange target.
2. Check that the centre point is sensible.
3. Check that the target centre lies inside the safe region in the default world.
4. Only then allow the robot-motion stage to proceed.

Part 4: Connect Vision to the Provided Grasp

Now connect the perception result to the supplied motion function. The key engineering principle is: perception should decide whether action is permitted; the motion component should execute the action.

Think of the control logic as a gate

This is the structure you need to implement.

```
result = detect_orange_block(bgr)

if result is None:
    # NOT FOUND → remain still
else:
    u, v, mask, contour = result
    # check whether (u, v) is inside the safe ROI
```

```
    # only then call the supplied grasp helper
```

Perception result	Safety result	Expected behaviour
No valid contour	N/A	Do not move
Valid target	Outside ROI	Do not grasp
Valid target	Inside ROI	Start supplied grasp sequence

Why separate perception and motion?

The detector should answer: 'What do I see, and where is it?' The grasp helper should answer: 'How do I execute the already-calibrated motion?'

Separating these components makes debugging easier and prevents a perception error from being mixed with arm-control code.

What to look for in the console

- A successful run should report that the target was found.
- The target should be reported as inside the safe region.
- The supplied motion should progress through its expected stages, such as OPEN → APPROACH → CLOSE → LIFT.
- The orange target should be physically lifted.

Do not rewrite the arm controller: The workshop supplies the grasp geometry and motion. Implementing inverse kinematics here would distract from the learning objective: connecting computer vision to a safe robot action.

Checkpoint questions

- Why is a visual safety gate useful before executing a pre-programmed grasp?
- Why is the detector output better passed as structured data rather than printing only a message?
- What would happen if the detector returned a centre even when no valid contour existed?

Part 5: Test One Controlled Failure

A robot-vision system should demonstrate not only success but also safe behaviour when its assumptions are violated. In this controlled test, move the orange target far enough that its detected image centre is outside the white safe region.

Test procedure

1. Run the successful default trial first and save the evidence.
2. Move the orange target to a position that places its image centre outside the safe region.
3. Reset and run the simulation again.
4. Confirm that the controller reports the target is outside the safe region.
5. Confirm that the robot does not execute the grasp.
6. Restore the original target pose after the test.

Record evidence

Evidence	What it demonstrates
Detected centre from successful trial	Perception output
Detection image	Contour + centre + safe ROI
Lifted-target screenshot	Successful perception-to-action integration
Controlled-failure console message	Safety gate works

Engineering mindset: A controlled failure is valuable evidence. You are showing that the system can refuse to act when a required condition is not satisfied.

Appendix

Debugging Guide

When something fails, do not immediately change several parts of the controller. Trace the pipeline in order.

Layer	Question	Useful evidence
Camera	Am I receiving an image?	Resolution / saved RGB image
Colour	Is orange isolated?	Mask image
Shape	Is the correct region selected?	Contour overlay
Geometry	Is the centre plausible?	Printed “(u,v)” + centre marker
Safety	Does the gate behave correctly?	ROI decision
Motion	Does the supplied grasp execute?	Console states + Webots view

A useful debugging sequence

1. Run the starter controller.
2. Test the camera alone.
3. Inspect the HSV mask.
4. Print the number of contours.
5. Print the selected contour area.
6. Print the centroid.
7. Draw the centroid and contour.
8. Test the safe ROI without calling the grasp.
9. Only after the gate works, enable the grasp.

Typical Python/OpenCV questions

Question	Short answer
Why “np.ones((5,5), np.uint8)”?	Creates a 5×5 unsigned-integer kernel for morphology.
Why “bgr.copy()”?	Creates an independent image for drawing annotations.
Why “max(..., key=cv2.contourArea)”?	Selects the contour with the greatest area.
Why check “if not contours”?	A valid camera frame may contain no candidate.
Why check “m00”?	Avoid division by zero when computing a centroid.
Why return “None”?	It cleanly represents 'no valid detection' to the main program.
Why use “// 2” for image centre?	Integer division gives a pixel coordinate.
Why use “abs(...) <= threshold”?	It tests whether the point lies within a symmetric distance from the centre.

Optional Extensions

These are not required for the core workshop. Use them only after you have a reliable detector, safety gate and grasp.

RangeFinder depth

The world includes a RangeFinder. A useful extension is to extract the depth values corresponding to the detected target mask and report a robust statistic such as the median depth.

Focus on the data-selection idea; consult the Webots RangeFinder documentation for the exact API.

```
# Conceptual workflow
depth_image = ...
target_depths = depth_image[mask > 0]
median_depth = np.median(target_depths)
```

Estimate target orientation

```
rect = cv2.minAreaRect(contour)
```

"cv2.minAreaRect()" returns a rotated rectangle describing the contour. This can provide an estimate of orientation. Remember that the angle convention needs interpretation and is not automatically a physical robot joint angle.

Do not over-engineer: The core workshop explicitly excludes inverse kinematics, camera calibration, 3D reconstruction and grasp planning. A strong implementation that is simple, testable and well explained is better than unnecessary complexity.

Using AI Responsibly

AI tools can be useful for learning unfamiliar functions, debugging error messages and explaining concepts.

They should support your understanding rather than replace it.

- Ask: 'Explain cv2.findContours and what RETR_EXTERNAL means.'
- Ask: 'Why might cv2.moments produce a zero m00?'
- Ask AI to explain an error message and suggest what to inspect.
- Ask for a small example of morphological opening/closing, then test your understanding in your own controller.
- Do not ask for or paste a complete final workshop solution as a substitute for your own implementation.
- Be able to explain every line of code you submit.

After using AI, try to describe the pipeline yourself: image → HSV → mask → morphology → contour → area → centroid → safety → grasp.

Submission and Evidence Checklist

The workshop asks you to submit your completed controller, a short result document, and optional discussion answers.

Item	Include
Controller	Completed "week9_starter.py", including your "detect_orange_block()" implementation
Successful result	Saved detection image and one screenshot showing the lifted target
Detection result	Detected centre for the successful default trial
Failure result	Console message from the controlled failure trial
Discussion	Answers to the workshop discussion questions, if required

Before you submit

- Your detector handles no-contour cases safely.
- Your detector rejects candidates below the specified area threshold.
- Your centroid calculation checks the moment denominator.
- Your contour and centre are visually correct.
- The safe ROI prevents an unsafe grasp attempt.
- The supplied grasp sequence still works.
- You tested both success and controlled failure.
- Your code is readable enough that you can explain it.

Quick Reference

Task	Remember
Get camera	“robot.getDevice("camera")”
Enable camera	“camera.enable(timestep)”
BGR → HSV	“cv2.cvtColor(bgr, cv2.COLOR_BGR2HSV)”
Threshold colour	“cv2.inRange(hsv, lower, upper)”
Create kernel	“np.ones((5,5), np.uint8)”
Morphology	“cv2.morphologyEx(mask, operation, kernel)”
Contours	“cv2.findContours(mask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)”
Area	“cv2.contourArea(contour)”
Moments	“cv2.moments(contour)”
Draw contour	“cv2.drawContours(...)”
Draw centre	“cv2.circle(...)”
Draw ROI	“cv2.rectangle(...)”
Save image	“cv2.imwrite(filename, image)”
Safe action	Only call the supplied grasp helper after a valid detection passes the ROI check

Workshop 9 is a compact example of robotics integration: perception produces evidence, a safety rule evaluates that evidence, and a tested motion component acts only when the conditions are acceptable.

Summary

The most important lesson is not a particular OpenCV function.

It is the engineering structure: convert raw sensor data into a meaningful representation, validate the result, and only then allow a physical action.

In this workshop, the chain is:

RGB camera → orange mask → cleaned mask → contour → centroid → safe ROI → provided grasp

That same perception-decision-action pattern appears repeatedly in real robotics systems, even when the sensors, algorithms and robot behaviours become much more sophisticated.